Entrega 1 - Implementación

Grupo 7

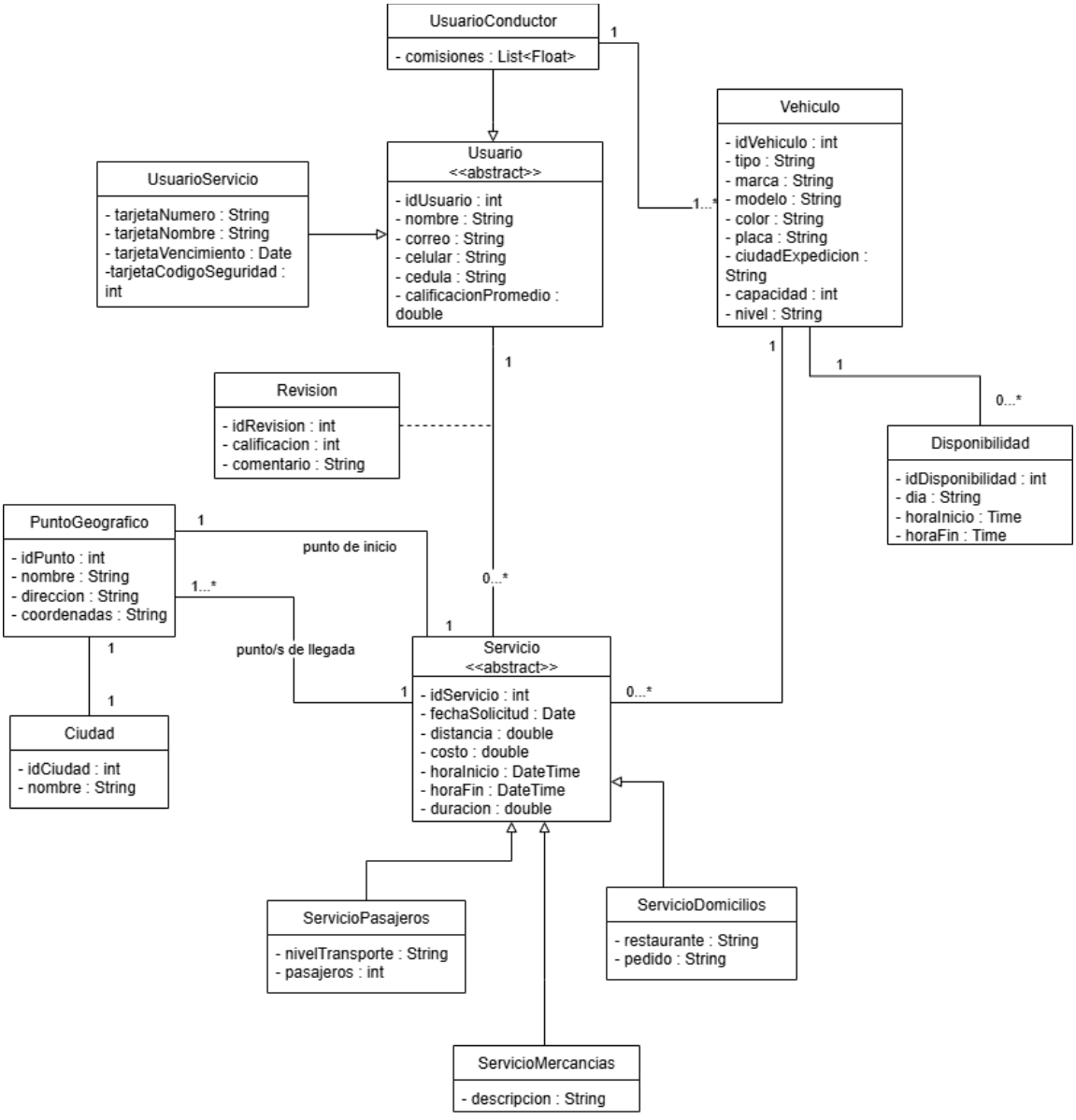
Juliana Vera Quintero – 202411275

Mateo Jaimes Rincon- 202411187

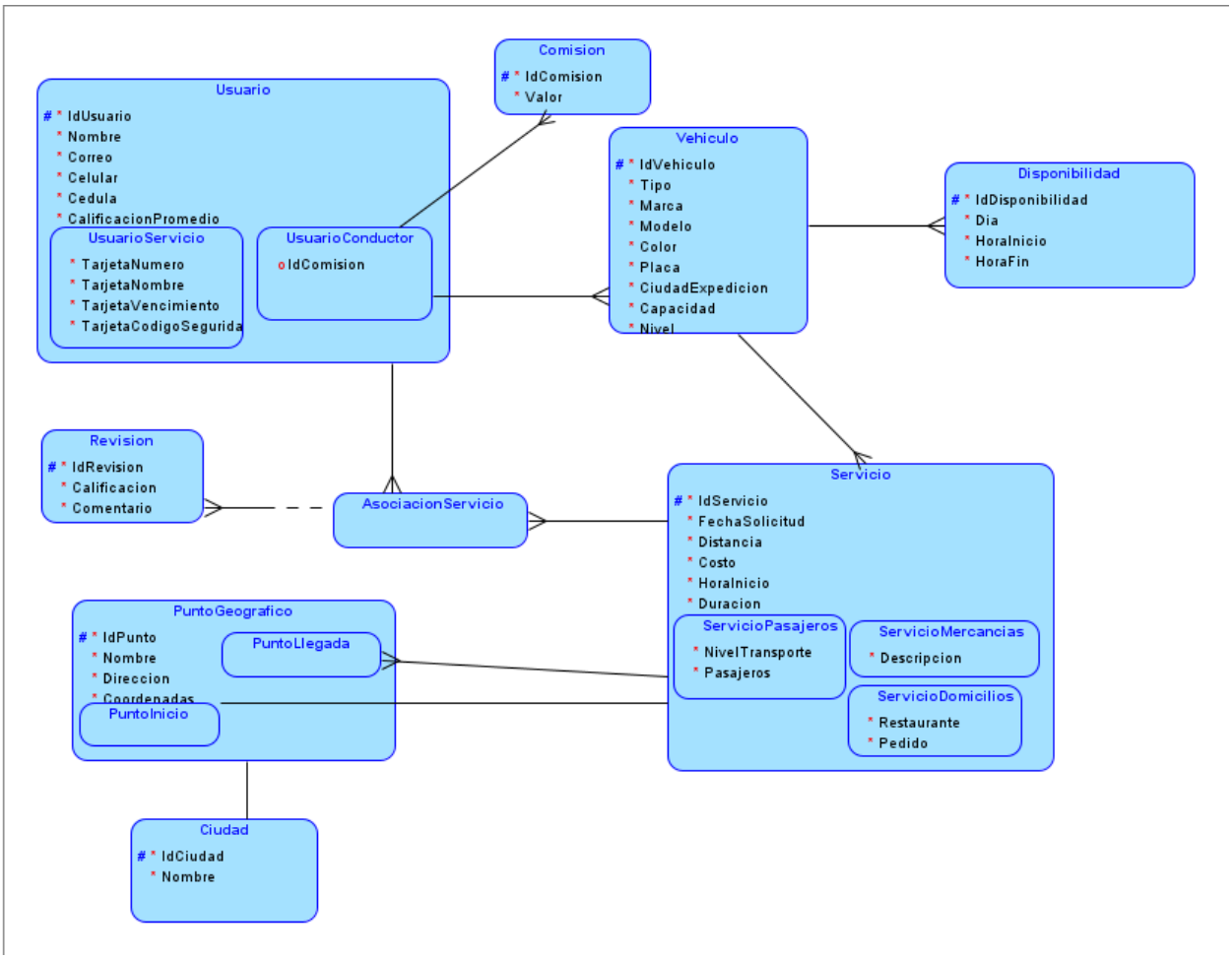
Daniel Andrés Gómez Ávila - 202413493

Análisis y modelo conceptual

Modelo conceptual en UML



Modelo conceptual en E/R



Modelo de datos relacional

Usuario					
idUsuario	**Nombre**	**Correo**	**Celular**	**Cedula**	**CalificacionPromedio**
PK, SA	NN	NN, ND	NN,ND	NN, ND, NC	$CK_{[0..5]}$

Es la tabla padre de las entidades de Usuario. Incluye los atributos base de todo usuario en la base de datos. Su llave primaria es el idUsuario. En el diagrama UML aparece como una superclase abstracta, heredando a dos clases. Por tanto, optamos por una de las alternativas para representar herencia según el algoritmo modificado de Chen, que era crear una tabla para cada una de las entidades y hacer referencia a la entidad padre por medio de llaves foráneas.

UsuarioConductor

idUsuario
PK, FK$_{Usuario.idUsuario}$

UsuarioServicio

idUsuario	TarjetaNumero	TarjetaNombre	TarjetaVencimiento	TarjetaCodigoSeguridad
PK, FK$_{Usuario.idUsuario}$	NN	NN	NN	NN

UsuarioConductor y UsuarioServicio son las especializaciones de la superclase abstracta Usuario, que en el diagrama de clases UML se representaba como una herencia. Se implementaron como tablas separadas, cada una con una PK que es también una llave foránea haciendo referencia a idUsuario. Se diferencian entre sí por sus funciones.

ComisiónConductor

idComision	idUsuarioConductor	Monto
PK, SA	FK$_{UsuarioConductor.idUsuario}$	NN

En el diagrama de clases UML, UsuarioConductor tiene un atributo llamado comisiones, que es una lista de datos numéricos tipo float. Sin embargo, según las reglas de normalización, los atributos en un modelo relacional deben ser atómicos. Por tanto, la solución planteada por el algoritmo es la creación de una tabla intermedia que represente esta relación. Tiene un id propio, una llave foránea que lo relaciona con su respectivo conductor, y un monto que correspondería en el UML al float en la lista.

Vehiculo

idVehiculo	idConductor	Tipo	Marca	Modelo	Color	Placa	CiudadExpedicion	Capacidad	Nivel
PK, SA	FK$_{UsuarioConductor.idUsuario}$	NN	NN	NN	NN	NN, ND	NN	NN	NN

Vehículo se relaciona principalmente con UsuarioConductor. En el UML se representa su relación de uno a muchos (un conductor tiene varios carros – un carro tiene solo un conductor). Por ello, en la tabla se representa con una llave foránea hacia el id del conductor, además de un id propio para su identificación y demás atributos básicos.

DisponibilidadVehiculo

idDisponibilidad	idVehiculo	Dia	HoraInicio	HoraFin
PK, SA	FK$_{Vehiculo.idVehiculo}$	NN	NN	NN

Se relaciona directamente con vehículo. Una “disponibilidad” solo puede hacer referencia a un vehículo, pero éste puede tener varias “disponibilidades”. Entonces, cuenta con una llave foránea haciendo referencia al id del vehículo, y una PK de id propio.

Servicio									
idServicio	idUsuario	idVehiculo	idPuntoInicio	FechaSolicitud	Distancia	Costo	HoraInicio	HoraFin	Duracion
PK, SA	$FK_{Usuario.idUsuario}$	$FK_{Vehiculo.idVehiculo}$	$FK_{PuntoGeografico.idPunto}$	NN	NN	NN	NN	NN	NN

Al igual que Usuario, Servicio es una superclase abstracta en el diagrama de clases UML, de la que derivan los tipos de servicios disponibles en la plataforma. Cuenta con una llave primaria que es el id, y con tres llaves foráneas: IDs de usuario, vehículo y punto de inicio, ya que independiente del tipo de servicio que se contrate, siempre van a estar asociadas a estas tres entidades. Es importante resaltar el hecho de que se relaciona a Usuario, la superclase, con el fin de simplificar y abstraer al máximo la relación entre estas dos entidades grandes. Además, esta tabla concuerda con lo descrito en el UML: un vehículo y un usuario pueden tener 0 a muchos servicios, pero cada servicio solo un vehículo y solo un usuario; y un punto de inicio y un servicio corresponden de manera única. Más adelante se explicará la relación con punto de llegada.

ServicioPasajeros		
idServicio	NivelTransporte	Pasajeros
PK, $FK_{Servicio.idServicio}$	NN	NN

ServicioDomicilios		
idServicio	Restaurante	Pedido
PK, $FK_{Servicio.idServicio}$	NN	NN

ServicioMercancias	
idServicio	descripcion
PK, $FK_{Servicio.idServicio}$	NN

Pasajeros, Domicilios y Mercancías son las tres especializaciones de la superclase abstracta Servicio, representado en el diagrama de clases UML como una herencia. En este modelo, al igual que para Usuario, la herencia se representa por medio de llaves foráneas que apuntan hacia el ID de Servicio en general, siendo a la vez la PK de cada una.

Ciudad		
idCiudad	Nombre	Punto
PK, SA	NN	$FK_{PuntoGeografico.idPunto}$

Ciudad es una entidad que solamente se relaciona con punto geográfico, y por tanto, es la única llave foránea que tiene: una ciudad tiene muchos puntos geográficos, que según el servicio serán de inicio o de llegada, pero un punto geográfico solo existe en una ciudad. Se identifica a una ciudad por medio de su PK que es ID único.

PuntoGeografico				
idPunto	**Nombre**	**Direccion**	**Coordenadas**	**Ciudad**
PK, SA	NN	NN	NN	$FK_{Ciudad.idCiudad}$

Punto geográfico puede representar punto de inicio o de llegada en un servicio determinado. Tiene su propio ID, que es su llave primaria, y una llave foránea que hace referencia a la ciudad en que se ubica.

PuntoLlegada	
idServicio	**idPunto**
PK, $FK_{Servicio.idServicio}$	PK, $FK_{PuntoGeografico.idPunto}$

Establecer la relación de Servicio con PuntoLlegada es más complejo que con PuntoInicio, ya que un servicio puede tener solo un punto de inicio pero puede tener varios puntos de llegada (paradas) durante el recorrido. Siguiendo una vez más el principio de atomicidad según las reglas de normalización, la solución propuesta por el algoritmo modificado de Chen es crear una tabla aparte que se relacione con las otras entidades, en este caso, PuntoLlegada. Tiene dos llaves foráneas que a su vez conforman la llave primaria: idServicio, y idPunto, ya que deben ser puntos geográficos y estar relacionados a un servicio.

AsociacionServicio		
idAsociacionServicio	**Usuario**	**Servicio**
PK, SA	$FK_{Usuario.idUsuario}$	$FK_{Servicio.idServicio}$

La relación entre Usuario y Servicio, las dos clases centrales y abstractas del proyecto, necesita ser descrita con una tabla adicional para facilitar la implementación y comprensión. Por ello, inventamos la entidad AsociacionServicio, que representa la base de toda posible interacción en la plataforma, con dos llaves foráneas haciendo referencia a Usuario y Servicio, y su propio ID siendo la llave primaria para identificarla.

Revision			
idRevision	**idAsociacionServicio**	**Calificacion**	**Comentario**
PK, SA	$FK_{AsociacionServicio.idAsociacionServicio}$	NN, $CK_{[0..5]}$	NN

En el diagrama UML, revisión está conectada a la relación entre Usuario y Servicio, y como fue explicado anteriormente, esta relación es descrita por una tabla. Así, la llave foránea que debe tener revisión hace referencia a AsociacionServicio, y tiene sentido porque una retroalimentación se hace respecto al servicio recibido por parte de un usuario. Tiene su propio ID que es PK, y dentro de sus atributos básicos, se hace CK de que calificación sea un valor numérico entre 0 y 5.

Justificación de normalización

Usuario: Todos los datos son atómicos y no hay campos con más de un valor (1FN). La llave primaria es simple (idUsuario), entonces no existen dependencias parciales (2FN). Todos los atributos dependen únicamente de la llave primaria; correo, celular y cédula son únicos pero dependen directamente de la PK (3FN). Por último, no hay dependencias funcionales donde un atributo no clave determine uno clave, y cada determinante es candidata (BCNF).

idUsuario → {Nombre,Correo,Celular,Cedula,CalificacionPromedio}

UsuarioConductor: No hay datos repetidos (1FN), la llave primaria es simple (2FN), y no hay dependencias transitivas (3FN), ya que representa una extensión lógica de Usuario (BCNF).

idUsuario → idUsuario

ComisiónConductor: Sus datos son atómicos (1FN), tiene una llave primaria simple sin dependencias parciales (2FN), el atributo Monto depende solamente de la llave primaria (3FN), y todos los determinantes son llaves candidatas (BCNF).

idComision → {idUsuarioConductor,Monto}

UsuarioServicio: Cada atributo es atómico (1FN), la llave primaria ID es simple sin dependencias parciales (2FN), toda la información de la tarjeta depende directamente de la llave primaria ID (3FN), y no hay atributos clave que determinen no clave (BCNF).

idUsuario→{TarjetaNumero,TarjetaNombre,TarjetaVencimiento,TarjetaCodigoSeguridad}

Vehiculo: Todos los atributos son atómicos, teniendo un vehículo por fila (1FN), la llave primaria es simple y no tiene dependencias parciales (2FN), ningún atributo no clave depende de uno clave (3FN). Placa podría ser una candidata única, pero no viola BCNF porque no determina atributos por fuera de la llave primaria.

idVehiculo →
{idConductor,Tipo,Marca,Modelo,Color,Placa,CiudadExpedicion,Capacidad,Nivel}

DisponibilidadVehiculo: Tiene los horarios separados por fila (1FN), la llave primaria es simple sin dependencias parciales (2FN), no tiene dependencias transitivas (3FN) y todo determinante es llave candidata, cumpliendo así con la forma BCNF.

idDisponibilidad → {idVehiculo,Dia,HoraInicio,HoraFin}

Servicio: Tiene solamente la información de un servicio por fila (1FN), tiene una llave primaria simple sin dependencias parciales (2FN), ningún atributo no clave determina otro (3FN) y todo depende únicamente de la llave primaria (BCNF).

idServicio → {idUsuario,idVehiculo,idPuntoInicio,FechaSolicitud,Distancia,Costo,HoraInicio,HoraFin,Duracion}

Ciudad: Todos los datos son atómicos (1FN), la llave primaria es simple (2FN), no hay dependencias transitivas (3FN) y la llave es el único determinante (BCNF).

idCiudad → {Nombre,Punto}

Revision: Cada revisión es independiente y atómica (1FN), la llave primaria es simple (2FN), no tiene dependencias transitivas (3FN) y todo determinante es clave (BCNF).

idRevision → {idEvaluador,idEvaluado,idServicio,Calificacion,Comentario}

ServicioPasajeros, ServicioDomicilios y ServicioMercancias: Cada tabla tiene ID_SERVICIO como llave primaria y atributos adicionales dependen únicamente de la PK, entonces cumple con todos los niveles de normalización hasta BCNF.

idServicio→{NivelTransporte,Pasajeros}, idServicio→{Restaurante,Pedido}, idServicio→Descripcion

PuntoGeografico: Sus atributos son atómicos (1FN), la llave primaria determina todo (2FN), no hay dependencias transitivas porque aunque ID_CIUDAD se relaciona con NOMBRE_CIUDAD, no rompe la tabla, la información de ciudad se normalizó, entonces cumple con 3FN y con BCNF.

idPunto → {Nombre,Direccion,Coordenadas,Ciudad}

PuntoLlegada: Los atributos son atómicos (1FN), no hay dependencia parcial que no sea PK completa, atributos no clave no determinan otros (3FN) y la llave primaria compuesta determina todo (BCNF).

{idServicio,idPunto} → idServicio,idPunto

Es posible determinar que el modelo relacional cumple en su totalidad con la forma normal de Boyce-Codd. Revisando tabla por tabla, todas cumplen con las tres formas normales: 1FN (atomicidad), 2FN (no dependencias parciales), 3FN (no dependencias transitivas), y finalmente BCNF. Esto garantiza que cada determinante de una relación es clave candidata, eliminando redundancias y asegurando que la información se organice de manera consistente y eficiente en todas las tablas del sistema.

Escenarios de prueba:

RF1 - REGISTRAR UNA CIUDAD: Registra una nueva ciudad en la que se prestarán servicios de transporte para ALPESCAB.

Escenario de prueba	Datos de entrada	Resultado esperado
Inserción exitosa	Quiero registrar una ciudad con idCiudad=1 y nombre= “Bogotá”.	Ciudad registrada con id=1.
PK duplicada	Quiero registrar otra ciudad con idCiudad=1 pero nombre= “Medellín”.	Error: violación de PK.
Nombre duplicado (unicidad lógica)	Quiero registrar una ciudad distinta con nombre= “Bogotá”.	Error: ciudad ya existe.

RF2 - REGISTRAR UN USUARIO DE SERVICIOS: Un usuario puede darse de alta en la aplicación, usando los datos descritos en la descripción general del caso.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar un usuarioServicio con id=10, nombre= “Carlos Pérez”, correo=“carlos@mail.com” , celular= “3109876543”, cédula= “987654321”, calificación promedio=4.0, tarjetaNúmero= “1234567812345678”, tarjetaNombre= “Carlos Pérez”, tarjeta Vencimiento= “12/28”,	UsuarioServicio registrado con id=10.

	tarjetaCodigoSeguridad= “123”.	
PK duplicada	Quiero registrar otro usuarioServicio con id=10 pero con nombre distinto.	Error: violación de PK.
Correo duplicado	Quiero registrar un usuarioServicio con correo “carlos@mail.com” que ya está registrado.	Error: correo duplicado.
Calificación fuera de rango	Quiero registrar un usuarioServicio con calificación promedio=6 y datos de tarjeta válidos.	Error: violación de restricción CHECK (0–5).
Número de tarjeta inválido	Quiero registrar un usuarioServicio con tarjetaNúmero= “1234” (menos de 16 dígitos), tarjetaNombre= “Carlos Pérez”, tarjetaVencimiento= “12/28”, tarjetaCodigoSeguridad= “123”.	Error: número de tarjeta inválido.
Fecha de vencimiento inválida	Quiero registrar un usuarioServicio con tarjetaVencimiento= “01/20” (fecha pasada).	Error: tarjeta vencida.
CVV inválido	Quiero registrar un usuarioServicio con tarjetaCodigoSeguridad= “12” (menos de 3 dígitos).	Error: CVV inválido.

RF3 - REGISTRAR UN USUARIO CONDUCTOR: Un usuario que va a prestar servicios de transporte puede darse de alta en la aplicación, usando los datos descritos.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar un usuario con id=20, nombre= “Carlos Pérez”, correo=“carlos@mail.com” , celular= “3109876543”, cédula= “987654321”, calificación promedio=4.2.	Usuario registrado con id=20.

PK duplicada	Quiero registrar otro usuario con id=20 pero nombre distinto.	Error: violación de PK.
Correo duplicado	Quiero registrar un usuario con correo "carlos@mail.com" que ya está registrado.	Error: correo duplicado.
Calificación fuera de rango	Quiero registrar un usuario con calificación promedio=6.	Error: violación de restricción CHECK (0–5).
Celular inválido	Quiero registrar un usuario con celular= "123".	Error: celular no cumple con formato requerido.
Cédula duplicada	Quiero registrar un usuario con cédula= "987654321" que ya está registrada.	Error: cédula duplicada.

RF4 - REGISTRAR UN VEHÍCULO PARA UN USUARIO CONDUCTOR Un conductor puede registrar un vehículo a su nombre con los datos indicados en la descripción generaldel caso.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar un vehículo con idVehiculo=100, tipo= "Automóvil", marca= "Toyota", modelo= "Corolla", color= "Blanco", placa= "ABC123", ciudadExpedición= "Bogotá", capacidad=4, nivel = "Estándar" y asignarlo al conductor id=20.	Vehículo registrado correctamente.
FK inválida	Quiero registrar un vehículo para el conductor id=99 que no existe.	Error: FK no encontrada.
Placa duplicada	Quiero registrar un vehículo con la misma placa= "ABC123".	Error: violación de unicidad.
Capacidad inválida	Quiero registrar un vehículo con capacidad=–3.	Error: fuera de rango.

RF5 - REGISTRAR LA DISPONIBILIDAD DE UN USUARIO CONDUCTOR Y SU VEHÍCULO PARA UN SERVICIO. Un conductor puede seleccionar días y rangos horarios en los cuales

prestará diferentes tipos de servicios. (transporte de pasajeros, entrega de comida, transporte de mercancías. En caso de prestar el servicio de transporte de pasajeros, el modelo y capacidad del vehículo determinarán el nivel de este (Estándar,Confort y Large). No es posible tener disponibilidades que se superpongan para un mismo conductor.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar disponibilidad con idDisponibilidad=1, vehículo id=100, día="lunes", horaInicio=08:00 y horaFin=18:00.	Disponibilidad registrada.
FK inválida	Quiero registrar disponibilidad para vehículo id=999 que no existe.	Error: FK no encontrada.
Horario inválido	Quiero registrar disponibilidad con horaFin=07:00 menor a horaInicio=08:00.	Error: restricción lógica.
Disponibilidad superpuesta	Quiero registrar disponibilidad de 10:00–12:00 el mismo día donde ya existe otra de 09:00–11:00.	Error: no se permiten rangos superpuestos.

RF6 - MODIFICAR LA DISPONIBILIDAD DE UN VEHÍCULO PARA SERVICIOS Un conductor puede modificar la disponibilidad en términos de días y horarios para las que prestará un servicio con su vehículo. No debe ser posible cambiar la disponibilidad si esta se superpone con otras disponibilidades del mismo conductor.

Escenario de prueba	Datos de entrada	Resultado esperado
Modificación exitosa	Quiero modificar la disponibilidad id=1 y cambiar horaFin a 20:00.	Disponibilidad actualizada.
Disponibilidad inexistente	Quiero modificar disponibilidad id=99.	Error: registro no encontrado.
Superposición inválida	Quiero modificar disponibilidad id=1 para que se traslape con otra disponibilidad existente, suponiendo que registre	Error: no se permiten rangos superpuestos.

	otra disponibilidad en el mismo horario.	

RF7 - REGISTRAR UN PUNTO GEOGRÁFICO Todos los usuarios pueden registrar puntos geográficos en cualquier momento. Esto requiere indicar unnombre (para establecimientos particulares), una dirección, unas coordenadas geográficas y una ciudad.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar un punto geográfico con idPunto=50, nombre= “Titan Plaza”, dirección= “Av. Boyacá #80-94, Bogotá”, coordenadas= “4.701, -74.042”, ciudad= “Bogotá”.	Punto geográfico registrado.
FK inválida	Quiero registrar un punto geográfico en ciudad id=99 que no existe.	Error: FK no encontrada.
Duplicado	Quiero registrar otro punto con el mismo nombre, dirección y ciudad.	Error: punto ya existe.

RF8 - SOLICITAR UN SERVICIO POR PARTE DE UN USUARIO DE SERVICIOS Un usuario puede solicitar un servicio particular, indicando un punto de partida y un punto de llegada. La aplicación calcula la tarifa si existen diferentes niveles, realiza un cobro al usuario y automáticamente le asigna un conductor que tenga disponibilidad, que se encuentre cerca y que no esté ya asignado a un servicio en este momento.

Escenario de prueba	Datos de entrada	Resultado esperado
Registro exitoso	Quiero registrar un servicio con idServicio=200, idUsuario=10, idVehiculo=100, idPuntoInicio=50, FechaSolicitud=“2025-08-24”, Distancia=12.5 km, Costo=25,000, HoraInicio=“08:00”, HoraFin=“08:25”, Duración=25 minutos.	Servicio registrado con idServicio=200.
PK duplicada	Quiero registrar un servicio con idServicio=200 (ya	Error: violación de PK.

	existente) pero distinto idUsuario.	
Usuario inexistente	Quiero registrar un servicio con idUsuario=999 (usuario no registrado en el sistema).	Error: usuario no encontrado.
Vehículo inexistente	Quiero registrar un servicio con idVehiculo=888 (vehículo no registrado).	Error: vehículo no encontrado.
Punto de inicio inexistente	Quiero registrar un servicio con idPuntoInicio=777 (punto no registrado en el sistema).	Error: punto de inicio no encontrado.
Distancia negativa	Quiero registrar un servicio con Distancia=–10 km.	Error: violación de restricción CHECK (Distancia > 0).
Costo negativo	Quiero registrar un servicio con Costo=–5000.	Error: violación de restricción CHECK (Costo > 0).
Hora fin anterior a hora inicio	Quiero registrar un servicio con HoraInicio="09:00" y HoraFin="08:30".	Error: HoraFin debe ser posterior a HoraInicio.
Duración inconsistente	Quiero registrar un servicio con HoraInicio="08:00", HoraFin="08:25" pero Duración=50 minutos.	Error: inconsistencia en cálculo de duración.

RF9 - REGISTRAR UN VIAJE PARA UN USUARIO DE SERVICIOS Y UN USUARIO CONDUCTOR Una vez un servicio ha sido prestado, el conductor se marca disponible nuevamente, y se registra en el histórico de conductor y pasajero los datos del viaje, incluyendo la hora de inicio y hora de finalización, la longitud, el costo y demás información relevante.

Escenario de prueba	Datos de entrada	Resultado esperado
Finalización exitosa del servicio	"Quiero finalizar el servicio con idServicio=200, idUsuario=10"	El sistema valida que el servicio existe, pertenece al usuario y vehículo correctos, está en curso. Registra la hora de fin, calcula la duración y actualiza el estado a

		Finalizado. Se guarda la asociación en AsociacionServicio(idAsociacionServicio=501, idUsuario=10, idServicio=200).
Servicio inexistente	"Quiero finalizar el servicio con idServicio=100"	El sistema valida y detecta que el servicio con idServicio=100 no existe en la BD. Retorna error: "El servicio no existe" y no realiza cambios en Servicio ni en AsociacionServicio.
Usuario no coincide	"Quiero finalizar el servicio con idServicio=100 pero con idUsuario=20"	El sistema valida que el servicio existe pero no corresponde al usuario ingresado. Retorna error: "El usuario no está asociado a este servicio". El registro en Servicio permanece sin cambios.
Servicio ya finalizado	"Quiero finalizar el servicio con idServicio=100, idUsuario=20, (cuando ya tiene HoraFin registrada)	El sistema valida que el servicio ya tiene una hora de fin registrada. Retorna error: "El servicio ya fue finalizado anteriormente". No se altera ningún dato en Servicio ni en AsociacionServicio.

RF10 - DEJAR UNA REVISIÓN POR PARTE DEL USUARIO DE SERVICIOS Un pasajero puede registrar una revisión a un conductor para un servicio prestado, dando una calificación y dejando un comentario de texto.

Escenario de prueba	Datos de entrada	Resultado esperado
Inserción exitosa	Quiero dejar una revisión el conductor que encontrare a traves de idAsociacionServicio=501, con calificación=5 y	Revisión registrada.

	comentario= “Excelente servicio”.	
FK inválida (servicio)	Quiero dejar revisión para idAsociacionServicio =999 inexistente.	Error: servicio no encontrado.
Calificación inválida	Quiero dejar revisión con calificación=7.	Error: fuera de rango (0–5).

RF11 - DEJAR UNA REVISIÓN POR PARTE DE UN USUARIO CONDUCTOR Un conductor puede registrar una revisión a un cliente para un servicio prestado, dando una calificación y dejando un comentario de texto.

Escenario de prueba	**Datos de entrada (lenguaje natural)**	**Resultado esperado**
Inserción exitosa	Quiero dejar una revisión al usuario que lo encontrare a través de idAsociacionServicio=501, con calificación=4 y comentario=“Buen cliente”.	Revisión registrada.
FK inválida (usuario)	Quiero dejar revisión para usuario id=999 que no existe.	Error: usuario no encontrado.
Calificación inválida	Quiero dejar revisión con calificación=–1.	Error: fuera de rango (0–5).

RFC1 - CONSULTAR EL HISTÓRICO DE TODOS LOS SERVICIOS PEDIDOS POR UN USUARIO Dado un usuario de servicios específico, se debe mostrar una lista de todos los servicios junto con toda su información relevante.

Escenario de prueba	**Datos de entrada**	**Resultado esperado**
Usuario con histórico de servicios	UsuarioServicio = “10101010”.	Lista con todos los servicios (fecha, partida, llegada, duración, costo, conductor).
Usuario sin servicios previos	UsuarioServicio = “30303030”.	Mensaje: “El usuario no tiene servicios registrados”.
Usuario inexistente	UsuarioServicio = “99999999”.	Error: “Usuario no encontrado en la base de datos”.

RFC2 - MOSTRAR LOS 20 USUARIOS CONDUCTORES QUE MÁS SERVICIOS HAN PRESTADO EN LA APLICACIÓN Esta consulta permite identificar qué conductores han prestado la mayor cantidad de servicios en la aplicación, indicando además de los datos del conductor, el número de servicios que han prestado.

Escenario de prueba	Datos de entrada	Resultado esperado
Más de 20 conductores con servicios	— (como es una consulta general de un listado de usuarios no requiere datos de entrada, solo muestra los conductores con más servicios de toda la lista)	Lista con los 20 conductores con mayor número de servicios, ordenados descendentemente.
Menos de 20 conductores con servicios	— (como es una consulta general de un listado de usuarios no requiere datos de entrada, solo muestra los conductores con más servicios de toda la lista)	Lista con todos los conductores disponibles, ordenados descendentemente.
Ningún conductor con servicios registrados	— (como es una consulta general de un listado de usuarios no requiere datos de entrada, solo muestra los conductores con más servicios de toda la lista)	Mensaje: “No existen servicios registrados por conductores”.

RFC3 - MOSTRAR EL TOTAL DE DINERO OBTENIDO POR USUARIOS CONDUCTORES PARA CADA UNO DE SUS VEHÍCULOS, DISCRIMINADO POR SERVICIOS Esta consulta permite visualizar a un conductor el dinero ganado por todos los servicios prestados. Acá debe aparecer un agregado por los servicios diferentes y el total de dinero obtenido (tras deducir las comisiones que cobra ALPESCAB) .

Escenario de prueba	Datos de entrada	Resultado esperado
Conductor con varios vehículos y servicios	ConductorID = “20202020”.	Reporte con filas: vehículo, tipo servicio, total ganado, total general.
Conductor con un solo vehículo y un solo servicio	ConductorID = “30303030”.	Reporte con una fila que muestre los valores correspondientes.

Conductor sin servicios registrados	ConductorID = “40404040”.	Mensaje: “El conductor no tiene servicios registrados”.
Conductor inexistente	ConductorID = “99999999”.	Error: “Conductor no encontrado en la base de datos”.

RFC4 - MOSTRAR LA UTILIZACIÓN DE SERVICIOS DE ALPESCAB EN UNA CIUDAD DURANTE UN RANGO DE FECHAS INDICADO. Esta consulta permite visualizar las estadísticas de uso de todos los servicios y niveles de estos para un rango de fechas inicial y final, junto con el porcentaje del total al que corresponde el número de servicios. Esto se muestra siempre comenzando por el servicio más usado al menos utilizado.

Escenario de prueba	**Datos de entrada**	**Resultado esperado**
Servicios en Bogotá en rango válido	Ciudad = “Bogotá”, FechaInicial = “2025-01-01”, FechaFinal = “2025-03-31”.	Reporte ordenado con servicios y niveles más usados al menos usados, con % de participación.
Servicios en ciudad sin registros	Ciudad = “Medellín”, FechaInicial = “2025-01-01”, FechaFinal = “2025-03-31”.	Mensaje: “No existen servicios registrados en Medellín para el rango de fechas”.
Rango de fechas sin servicios	Ciudad = “Bogotá”, FechaInicial = “2024-01-01”, FechaFinal = “2024-01-31”.	Mensaje: “No se encontraron servicios en el rango de fechas indicado”.
Ciudad inexistente en la BD	Ciudad = “Barrancabermeja”.	Error: “Ciudad no encontrada en la base de datos”.

Implementación del esquema de la base de datos

Diseño de sentencias SQL

RFC1: Consultar el histórico de todos los servicios pedidos por un usuario

```sql
-- Requerimiento funcional de consulta 1

SELECT S.ID_SERVICIO,
       S.FECHA_SOLICITUD,
       S.TIPO_SERVICIO,
       S.ESTADO,
       S.PRECIO_TOTAL,
       U.NOMBRE AS NOMBRE_USUARIO,
       U.CORREO,
       U.CELULAR
FROM SERVICIO S
INNER JOIN USUARIO U
    ON S.ID_USUARIO_SERVICIO = U.ID_USUARIO
WHERE U.ID_USUARIO = 5;
```

Se usan las tablas de Servicio y Usuario, la primera con sus atributos de id, fecha de solicitud, tipo, estado, precio total, y la segunda con nombre, correo y celular. Utilizamos un inner join porque solo necesitamos los servicios que le pertenecen a ese usuario, filtramos con where y finalmente mostramos datos tanto del servicio como del usuario.

RFC2: Mostrar los 20 usuarios conductores que más servicios han prestado en la aplicación

```sql
-- Requerimiento funcional de consulta 2

SELECT U.ID_USUARIO,
       U.NOMBRE,
       U.CORREO,
       COUNT(S.ID_SERVICIO) AS TOTAL_SERVICIOS
FROM USUARIO U
INNER JOIN USUARIO_CONDUCTOR UC ON U.ID_USUARIO = UC.ID_USUARIO
INNER JOIN VEHICULO V ON UC.ID_USUARIO = V.ID_CONDUCTOR
INNER JOIN SERVICIO S ON V.ID_VEHICULO = S.ID_VEHICULO
GROUP BY U.ID_USUARIO, U.NOMBRE, U.CORREO
ORDER BY TOTAL_SERVICIOS DESC
FETCH FIRST 20 ROWS ONLY;
```

Usamos las tablas de usuario, usuario conductor, vehículo y servicio, específicamente los atributos de id, nombre y correo. Realizamos un inner join porque solo necesitamos usuarios que sean conductores y tengan servicios, luego se cuentan los servicios por conductor, se agrupan los resultados, y se ordena para luego retornar los 20 primeros.

RFC3: Mostrar el total de dinero obtenido por usuarios conductores para cada uno de sus vehículos, discriminado por servicios

```sql
-- Requerimiento funcional de consulta 3

SELECT V.ID_VEHICULO,
       V.PLACA,
       V.MODELO,
       COUNT(S.ID_SERVICIO) AS TOTAL_SERVICIOS,
       SUM(S.COSTO - (S.COSTO * 0.10)) AS TOTAL_GANADO
FROM VEHICULO V
INNER JOIN SERVICIO S ON V.ID_VEHICULO = S.ID_VEHICULO
WHERE V.ID_CONDUCTOR = 10
GROUP BY V.ID_VEHICULO, V.PLACA, V.MODELO
ORDER BY TOTAL_GANADO DESC;
```

Utilizamos las tablas de vehículo y servicio, ya que tienen los atributos de identificación del vehículo y la información del costo del servicio. Realizamos un inner join porque queremos encontrar únicamente los servicios asociados a un vehículo, y con este método logramos hallar esas coincidencias. Luego filtramos con where, reemplazando el 10 por el id del conductor deseado, para finalmente agrupar y ordenar de manera descendente.

RFC4: Mostrar la utilización de servicios de AlpesCab en una ciudad durante un rango de fechas indicado

```
-- Requerimiento funcional de consulta 4

SELECT C.NOMBRE AS CIUDAD,
       COUNT(S.ID_SERVICIO) AS TOTAL_SERVICIOS
FROM SERVICIO S
INNER JOIN PUNTO_GEOGRAFICO P ON S.ID_PUNTO_INICIO = P.ID_PUNTO
INNER JOIN CIUDAD C ON P.ID_CIUDAD = C.ID_CIUDAD
WHERE C.NOMBRE = 'Bogotá'
  AND S.FECHA_SOLICITUD BETWEEN TO_DATE('2025-01-01', 'YYYY-MM-DD')
                            AND TO_DATE('2025-03-01', 'YYYY-MM-DD')
GROUP BY C.NOMBRE
ORDER BY TOTAL_SERVICIOS DESC;
```

Usamos las tablas de ciudad, servicio y punto geográfico, específicamente sus atributos de identificación. Necesitamos un inner join para correlacionar un punto geográfico con uno de inicio para un servicio, así como un punto con una ciudad. Luego filtramos con where por el nombre de la ciudad deseada, y las fechas en el rango de consulta, para finalmente agrupar por nombre de ciudad y ordenar de manera descendente.

Documentación del Script de Base de Datos

1. Borrado de tablas existentes

Antes de crear las tablas se eliminan todas las que puedan existir previamente en el esquema, para evitar conflictos de claves primarias, foráneas o restricciones de unicidad. Se usa DROP TABLE ... CASCADE CONSTRAINTS para garantizar que también se eliminen sus restricciones dependientes. Esto asegura que el script sea **idempotente**, es decir, se pueda ejecutar varias veces sin causar errores.

Script:

```
-- Eliminar tablas si existen para evitar errores al crear
DROP TABLE ASOCIACION_SERVICIO CASCADE CONSTRAINTS;
DROP TABLE PUNTO_LLEGADA CASCADE CONSTRAINTS;
DROP TABLE SERVICIO_MERCANCIAS CASCADE CONSTRAINTS;
DROP TABLE SERVICIO_DOMICILIOS CASCADE CONSTRAINTS;
DROP TABLE SERVICIO_PASAJEROS CASCADE CONSTRAINTS;
DROP TABLE REVISION CASCADE CONSTRAINTS;
DROP TABLE SERVICIO CASCADE CONSTRAINTS;
DROP TABLE DISPONIBILIDAD_VEHICULO CASCADE CONSTRAINTS;
DROP TABLE VEHICULO CASCADE CONSTRAINTS;
DROP TABLE COMISION_CONDUCTOR CASCADE CONSTRAINTS;
DROP TABLE USUARIO_CONDUCTOR CASCADE CONSTRAINTS;
DROP TABLE USUARIO_SERVICIO CASCADE CONSTRAINTS;
DROP TABLE PUNTO_GEOGRAFICO CASCADE CONSTRAINTS;
DROP TABLE CIUDAD CASCADE CONSTRAINTS;
DROP TABLE USUARIO CASCADE CONSTRAINTS;
```

2. Creación de tablas

En esta sección se definen todas las tablas del modelo relacional, siguiendo las entidades y relaciones del dominio:

- USUARIO, USUARIO_CONDUCTOR, USUARIO_SERVICIO → gestionan roles y datos personales.
- VEHICULO y DISPONIBILIDAD_VEHICULO → gestionan la flota y sus horarios.
- CIUDAD y PUNTO_GEOGRAFICO → definen ubicaciones.
- SERVICIO y sus especializaciones (PASAJEROS, DOMICILIOS, MERCANCIAS) → representan los servicios prestados.
- REVISION → almacena calificaciones y comentarios.
- COMISION_CONDUCTOR → gestiona pagos.
- ASOCIACION_SERVICIO y PUNTO_LLEGADA → gestionan relaciones adicionales.

Cada tabla tiene su clave primaria y restricciones de integridad (NOT NULL, UNIQUE, CHECK).

Script:

```
-- Crear tablas según el modelo relacional
CREATE TABLE USUARIO (
...
);
```

```
CREATE TABLE USUARIO_CONDUCTOR (
...
);

CREATE TABLE COMISION_CONDUCTOR (
...
);

CREATE TABLE USUARIO_SERVICIO (
...
);

CREATE TABLE VEHICULO (
...
);

CREATE TABLE DISPONIBILIDAD_VEHICULO (
...
);

CREATE TABLE CIUDAD (
...
);

CREATE TABLE PUNTO_GEOGRAFICO (
...
);

CREATE TABLE SERVICIO (
...
);

CREATE TABLE REVISION (
...
);

CREATE TABLE SERVICIO_PASAJEROS (
...
);

CREATE TABLE SERVICIO_DOMICILIOS (
...
);

CREATE TABLE SERVICIO_MERCANCIAS (
...
);

CREATE TABLE PUNTO_LLEGADA (
...
```

```
);

CREATE TABLE ASOCIACION_SERVICIO (
...
);
```

3. Definición de claves foráneas

Aquí se conectan las tablas entre sí mediante relaciones de integridad referencial, para garantizar la coherencia de los datos:

- Conductores - Usuarios
- Vehículos - Conductores
- Servicios - Usuarios, Vehículos, Puntos
- Revisiones - Usuarios y Servicios
- Especializaciones - Servicios
- Asociaciones - Usuarios y Servicios

Esto permite mantener la consistencia del modelo relacional.

Script:

```
-- Agregar claves foráneas
ALTER TABLE USUARIO_CONDUCTOR
ADD CONSTRAINT fk_usuario_conductor FOREIGN KEY (ID_USUARIO)
REFERENCES USUARIO(ID_USUARIO);

ALTER TABLE USUARIO_SERVICIO
ADD CONSTRAINT fk_usuario_servicio FOREIGN KEY (ID_USUARIO)
REFERENCES USUARIO(ID_USUARIO);

ALTER TABLE COMISION_CONDUCTOR
ADD CONSTRAINT fk_comision_conductor FOREIGN KEY (ID_USUARIO_CONDUCTOR)
REFERENCES USUARIO_CONDUCTOR(ID_USUARIO);

...

ALTER TABLE ASOCIACION_SERVICIO
ADD CONSTRAINT FK_ASOCIACION_SERVICIO_SERVICIO
FOREIGN KEY (SERVICIO) REFERENCES SERVICIO(ID_SERVICIO);
```

4. Escenarios de prueba iniciales

Se insertan manualmente los primeros datos de prueba para verificar que el modelo funciona:

- Una ciudad (Bogotá).
- Dos usuarios: un conductor y un pasajero.
- Un vehículo para el conductor.
- Un punto geográfico.
- Tres servicios iniciales: pasajero, domicilio y mercancía.
- Sus asociaciones, revisiones, comisiones, disponibilidad y puntos de llegada.

Estos inserts sirven como casos de prueba básicos para validar las tablas y restricciones.

Script:

```
-- Insertar ciudad
INSERT INTO CIUDAD (ID_CIUDAD, NOMBRE) VALUES (1, 'Bogotá');

-- Insertar usuarios
INSERT INTO USUARIO (...);
INSERT INTO USUARIO (...);

-- Marcar roles
INSERT INTO USUARIO_CONDUCTOR (...);
INSERT INTO USUARIO_SERVICIO (...);

-- Vehículo
INSERT INTO VEHICULO (...);

-- Punto geográfico
INSERT INTO PUNTO_GEOGRAFICO (...);

-- Servicios iniciales (viaje, domicilio, mercancía)
INSERT INTO SERVICIO (...);
INSERT INTO SERVICIO_PASAJEROS (...);
INSERT INTO SERVICIO_DOMICILIOS (...);
INSERT INTO SERVICIO_MERCANCIAS (...);

-- Asociaciones
INSERT INTO ASOCIACION_SERVICIO (...);

-- Revisiones
INSERT INTO REVISION (...);

-- Comisiones
```

```
INSERT INTO COMISION_CONDUCTOR (...);

-- Disponibilidad del vehículo
INSERT INTO DISPONIBILIDAD_VEHICULO (...);

-- Puntos de llegada
INSERT INTO PUNTO_LLEGADA (...);
```

5. Consultas de verificación

Se ejecutan SELECT * sobre todas las tablas para comprobar que los datos iniciales se insertaron correctamente.

Esto sirve como **verificación visual** de que las entidades y relaciones están funcionando.

Script:

```
SELECT * FROM ASOCIACION_SERVICIO;
SELECT * FROM CIUDAD;
SELECT * FROM COMISION_CONDUCTOR;
SELECT * FROM DISPONIBILIDAD_VEHICULO;
SELECT * FROM PUNTO_GEOGRAFICO;
SELECT * FROM PUNTO_LLEGADA;
SELECT * FROM REVISION;
SELECT * FROM SERVICIO;
SELECT * FROM SERVICIO_DOMICILIOS;
SELECT * FROM SERVICIO_MERCANCIAS;
SELECT * FROM SERVICIO_PASAJEROS;
SELECT * FROM USUARIO;
SELECT * FROM USUARIO_CONDUCTOR;
SELECT * FROM USUARIO_SERVICIO;
SELECT * FROM VEHICULO;
```

6. Población masiva de datos

Una vez validados los escenarios de prueba, se generan datos a gran escala para simular un ambiente realista:

- Ciudades y puntos geográficos (10 ciudades y 28 puntos).
- 100 conductores y 200 pasajeros, con roles asociados.
- 100 vehículos distribuidos entre los conductores.
- 1000 servicios (70% pasajeros, 20% domicilios, 10% mercancías).
- Asociaciones entre usuarios y servicios (incluyendo carpooling).
- Revisiones generadas de manera masiva.
- Comisiones de conductores calculadas.
- Disponibilidad de vehículos en turnos semanales.
- Puntos de llegada distribuidos en todos los servicios.

Script (fragmento representativo):

```
-- Insertar 100 conductores
INSERT INTO USUARIO (...)
SELECT ... FROM dual CONNECT BY LEVEL <= 100;

-- Insertar 200 pasajeros
INSERT INTO USUARIO (...)
SELECT ... FROM dual CONNECT BY LEVEL <= 200;

-- Insertar 100 vehículos
INSERT INTO VEHICULO (...)
SELECT ... FROM dual CONNECT BY LEVEL <= 100;

-- Insertar 1000 servicios
INSERT INTO SERVICIO (...)
SELECT ... FROM dual CONNECT BY LEVEL <= 1000;

-- Distribuir especializaciones
INSERT INTO SERVICIO_PASAJEROS (...);
INSERT INTO SERVICIO_DOMICILIOS (...);
INSERT INTO SERVICIO_MERCANCIAS (...);

-- Asociaciones masivas
INSERT INTO ASOCIACION_SERVICIO (...);

-- Revisiones masivas
INSERT INTO REVISION (...);

-- Comisiones masivas
INSERT INTO COMISION_CONDUCTOR (...);

-- Disponibilidades
INSERT INTO DISPONIBILIDAD_VEHICULO (...);

-- Puntos de llegada
INSERT INTO PUNTO_LLEGADA (...);
```

7. Nota final

Para la generación de los scripts de población masiva de las tablas (conductores, pasajeros, vehículos, servicios, asociaciones, revisiones, comisiones, disponibilidades y puntos de llegada) se utilizó apoyo de ChatGPT, lo cual permitió automatizar el diseño de inserciones repetitivas y garantizar la consistencia de las relaciones, de esta forma fue posible

simplificar el trabajo de inserción de datos uno por uno y realizar una especie de iteración que generaran la misma respuesta de forma más sencilla.

Desarrollo de la aplicación

1. Resumen / propósito

La aplicación es un backend construido con Spring Boot que gestiona usuarios (conductores y pasajeros), vehículos, servicios (viajes, domicilios, mercancías), revisiones, disponibilidad y relaciones entre usuarios y servicios. El objetivo de esta parte del proyecto es documentar la arquitectura, describir las clases Java (controladores, repositorios, entidades).

2. Arquitectura general

- Capa de presentación (REST controllers): Exponen endpoints HTTP (GET/POST/PUT/DELETE). Ej.: /usuarios, /servicios, /ciudades, etc.
- Capa de persistencia (repositories): Interfaces que realizan darX, insertarX, actualizarX, eliminarX.
- Capa de dominio (model/entidades): Clases que representan las tablas: Usuario, UsuarioConductor, UsuarioServicio, Vehiculo, Ciudad, PuntoGeografico, Servicio, ServicioPasajeros, ServicioDomicilios, ServicioMercancias, Revision, DisponibilidadVehiculo, ComisionConductor, PuntoLlegada, AsociacionServicio.
- Base de datos: Oracle con tablas y constraints definidas por el script SQL que ejecutaste. Inserts masivos generan datos para pruebas.

3. Controladores — descripción y endpoints principales

3.1 UsuarioController:

- GET /usuarios — lista usuarios
- GET /usuarios/{id} — obtener usuario
- POST /usuarios/new/save — crear usuario (body JSON)
- POST /usuarios/{id}/edit/save — actualizar usuario
- DELETE /usuarios/{id}/delete — eliminar usuario

Ejemplo JSON para crear usuario (Request Body):

```
{
 "nombre": "Pedro Pérez",
 "correo": "pedro.perez@mail.com",
 "celular": "3001234567",
 "cedula": "12345678",
 "calificacionPromedio": 4.5
}
```

Script SQL equivalente (insert simple):

```
INSERT INTO USUARIO (ID_USUARIO, NOMBRE, CORREO, CELULAR, CEDULA, CALIFICACION_PROMEDIO)
VALUES (10, 'Pedro Pérez', 'pedro.perez@mail.com', '3001234567', '12345678', 4.5);
```

3.2 UsuarioConductorController:

- GET /usuariosConductores
- GET /usuariosConductores/{id}
- POST /usuariosConductores/{idUsuario}/new/save — convierte usuario en conductor
- PUT /usuariosConductores/{idUsuario}/update
- DELETE /usuariosConductores/{idUsuario}/delete

SQL para convertir usuario en conductor:

```
INSERT INTO USUARIO_CONDUCTOR (ID_USUARIO) VALUES (10);
```

3.3 UsuarioServicioController

- GET /usuariosServicio
- GET /usuariosServicio/{id}
- POST /usuariosServicio/{idUsuario}/new/save — agrega tarjeta (usuario pasa a ser usuario-servicio)
- PUT y DELETE análogos.

SQL equivalente:

```
INSERT INTO USUARIO_SERVICIO (ID_USUARIO, TARJETA_NUMERO, TARJETA_NOMBRE, TARJETA_VENCIMIENTO, TARJETA_CODIGO_SEGURIDAD)
```

VALUES (11, '4111111111111111', 'Ana', TO_DATE('2027-12-31','YYYY-MM-DD'), '123');

3.4 VehiculoController:

- GET /vehiculos
- GET /vehiculos/{id}
- POST /vehiculos/{id}/new/save — JSON con conductor.idUsuario y demás campos
- POST /vehiculos/{id}/edit/save
- DELETE /vehiculos/{id}/delete

Ejemplo SQL insert vehículo:

INSERT INTO VEHICULO (ID_VEHICULO, ID_CONDUCTOR, TIPO, MARCA, MODELO, COLOR, PLACA, CIUDAD_EXPEDICION, CAPACIDAD, NIVEL)
VALUES (2, 10, 'Carro', 'Toyota', 'Corolla', 'Blanco', 'PL010', 'Bogotá', 4, 'Económico');

3.5 Ciudades y Puntos (CiudadController, PuntoGeograficoController):

- GET /ciudades,
- GET /ciudades/{id},
- POST /ciudades/new/save,
- DELETE /ciudades/{id}/delete
- GET /puntos-geograficos, POST /puntos-geograficos/new/save, etc.

SQL (ejemplo):

INSERT INTO CIUDAD (ID_CIUDAD, NOMBRE) VALUES (1,'Bogotá');
INSERT INTO PUNTO_GEOGRAFICO (ID_PUNTO, NOMBRE, DIRECCION, COORDENADAS, ID_CIUDAD)
VALUES (1,'Parque Central','Cra 7 # 12-34','4.60971,-74.08175',1);

3.6 Servicios y especializaciones (ServicioController, ServicioPasajerosController, ServicioDomiciliosController, ServicioMercanciasController):

- GET /servicios,
- GET /servicios/{id},
- POST /servicios (parametros con @RequestParam en controllers),
- PUT y DELETE.

Las especializaciones (servicios-pasajeros, servicios-domicilios, servicios-mercancias) usan sus propios endpoints para insertar filas en tablas especializadas.

SQL (ejemplo):

INSERT INTO SERVICIO (ID_SERVICIO, ID_USUARIO, ID_VEHICULO, ID_PUNTO_INICIO, FECHA_SOLICITUD, DISTANCIA, COSTO, HORA_INICIO, HORA_FIN, DURACION) VALUES (4, 103, 10, 2, TO_TIMESTAMP('2025-09-29 08:30:00','YYYY-MM-DD HH24:MI:SS'), 12.5, 25000, TO_TIMESTAMP('2025-09-29 08:35:00','YYYY-MM-DD HH24:MI:SS'), TO_TIMESTAMP('2025-09-29 08:55:00','YYYY-MM-DD HH24:MI:SS'), NUMTODSINTERVAL(20,'MINUTE'));
INSERT INTO SERVICIO_PASAJEROS (ID_SERVICIO, NIVEL_TRANSPORTE, PASAJEROS) VALUES (4, 'Económico', 2);

3.7 Revisiones, Disponibilidad, Puntos de llegada, Comisiones, Asociaciones:

Controllers: /revisiones, /disponibilidades-vehiculo, /puntos-llegada, /comisiones-conductor, /asociaciones-servicio.

Ejemplo SQL para revisión o asociación:

INSERT INTO REVISION (ID_REVISION, ID_EVALUADOR, ID_EVALUADO, ID_SERVICIO, CALIFICACION, COMENTARIO) VALUES (1, 103, 10, 4, 4.5, 'Buen servicio');

INSERT INTO ASOCIACION_SERVICIO (USUARIO, SERVICIO) VALUES (103, 4);

INSERT INTO DISPONIBILIDAD_VEHICULO (ID_DISPONIBILIDAD, ID_VEHICULO, DIA, HORA_INICIO, HORA_FIN) VALUES (6, 10, 'Lunes', TO_TIMESTAMP('2025-09-29 08:00:00','YYYY-MM-DD HH24:MI:SS'), TO_TIMESTAMP('2025-09-29 18:00:00','YYYY-MM-DD HH24:MI:SS'));

INSERT INTO COMISION_CONDUCTOR (ID_COMISION, ID_USUARIO_CONDUCTOR, MONTO) VALUES (101, 10, 5000);

4. Organización del código Java

El proyecto sigue una estructura simple y clara basada en Spring Boot:

Paquetes principales

- controller — endpoints REST: reciben peticiones, validan y llaman a los repositorios.

- repositorio — consultas y DML (muchas veces nativeQuery) que interactúan directamente con la BD Oracle.
- modelo — entidades JPA (@Entity) que representan las tablas.
- uniandes.edu.co.proyecto.ProyectoApplication — clase de arranque (@SpringBootApplication).

Responsabilidades

- Los controllers exponen la API y devuelven ResponseEntity.
- Los repositorios ejecutan SQL (insert/update/delete/select). Usan @Modifying/@Transactional para DML.
- Las entidades sirven para serializar JSON y mapear filas de BD.

Notas importantes

- Se usan queries nativas para respetar la estructura/constraints de Oracle (secuencias, TO_TIMESTAMP, NUMTODSINTERVAL, etc.).
- application.properties tiene spring.jpa.hibernate.ddl-auto=none porque las tablas se crean por script, no por Hibernate.
- Endpoints típicos: /usuarios, /vehiculos, /servicios, /ciudades, etc. Crear/editar devuelve JSON con confirmación y, cuando aplica, el id del nuevo recurso.